**Practicum opdracht RTLinux en Interfaces**

Ruben Oosterwijk – 500807881

## Real Time Operating

### Impact Linux RT patch – Raspberry PI 4

Voor mijn onderzoek naar de linux RealTime patch voor de Raspberry Pi 4, heb ik twee artikelen gevolgd.

1: <https://lemariva.com/blog/2019/09/raspberry-pi-4b-preempt-rt-kernel-419y-performance-test>

En

2: <https://lemariva.com//blog/2018/02/raspberry-pi-rt-preempt-vs-standard-kernel-4-14-y>

En als extra hulpbron: <https://medium.com/@patdhlk/realtime-linux-e97628b51d5d>

In deze artikelen wordt de Preempt-RT Patch voor de Kernel 4.14.y geïnstalleerd.

Daarna worden twee test uitgevoerd om de impact van de Real Time patch te onderzoeken.

In blog 1: worden de stappen beschreven die ik ook heb gebruikt om de patch te installeren.

Na het installeren heb ik de twee tests uitgevoerd vóór het herstarten van de pi.

De patch wordt pas in gang gezet bij het opnieuw opstarten van de pi. Dat komt door het toevoegen van de line : ‘kernel=kernel7\_rt.img’ in “/boot/config.txt”.

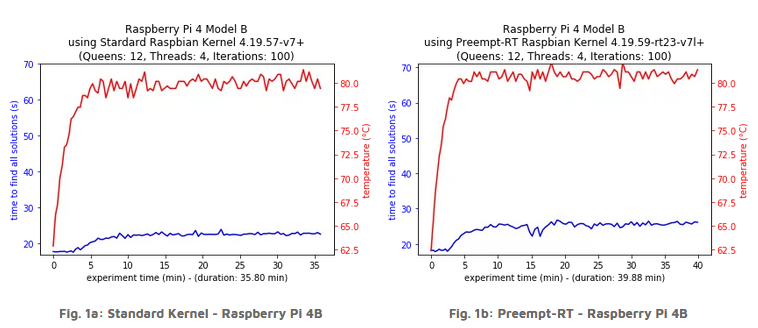
Dit geeft mij de opportuniteit om de test in de default staat uit te voeren en een zo de baseline kunnen zetten.

Ik heb de inbegrepen test gebruikt in blog 1: N-Queens-Problem: (zie: [uitleg](https://www.geeksforgeeks.org/n-queen-problem-backtracking-3/) en [source code](https://github.com/lemariva/N-Queens-Problem) en [resultaten](https://github.com/rubenoosterwijk/rtos_RTLinux/tree/main/N-Queens-Problem-results))

Verder heb ik ook de cyclic test uitgevoerd, beschreven in blog 2: (zie: [uitleg](https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start) en [source code](git://git.kernel.org/pub/scm/utils/rt-tests/rt-tests.git) en [resultaten](https://github.com/rubenoosterwijk/rtos_RTLinux/tree/main/rt-tests-results))

Hierbij ben ik tot de volgende conclusie gekomen:

N-Queens-Problem:



The Preempt-RT patched kernel was LANGZAMER om de N-queens-problems op te lossen dan de default kernel. De temperatuur was wel iets hoger.

Cyclictest resultaten:  
Default:

Afbeelding met tekst

Automatisch gegenereerde beschrijving

RT-Patch:

Afbeelding met tekst, binnen

Automatisch gegenereerde beschrijving

The maximale latency van de default kernel was 1632 us, terwijl de maximale latencie van de Real time kernal is 82 us. Dit bewijst dat de latency 19,9x zo geoptimaliseerd is.

Dit beantwoord ook de vraag of de detectie van het indrukken van het rempedaal binnen een 1msec wordt gedetecteerd. Waarbij het bij de standaard kernal niet die garantie kan geven (1000 us = 1ms). Wanneer de Real Time Patch kernel wordt gebruikt is de maximale latency onder de 1000 us.

### Serial Transfer – Raspberry PI 4

Voor het onderzoek naar de UART serial transfer rates heb ik het volgende artikel gebruikt:  
<https://fw.hardijzer.nl/?p=138>

Wanneer je PI koopt kan de UART chip op de pi data versturen op 115200 baud, samen met veel debug data op de serial paden.

De limiet wordt gecreëerd door de UART Clock. Die laat alleen baudrates toelaat van UART0\_CLOCK/16, en UART0\_CLOCK staat op 3Mhz als standaard. Dat de baudrate limiteerd op 187500baud. Om dit te verhogen moeten we de UART0\_CLOCK verhogen.

* In “/boot/config.txt” zetten we de volgende configuratie parameters:

“init\_uart\_clock=16000000”

* “/boot/start.elf” updaten naar de laatste versie: [laatste versie](https://github.com/raspberrypi/firmware/tree/master/boot)
* Nieuwe kernel compilen en installeren.

Dit laat toe om via de commando de baudrate hoger te zetten: “sudo stty -F /dev/ttyAMA0 1000000”

Met de commande kunnen een bericht (BOO) via de serial transfer versturen: “echo  BOO > /dev/ttyAMA0” en dit bericht kon ik dan via een FTDI device opvangen of op de pi zelf door de TX en RX pinnen te linken.

Voor de rest moeten we alle andere componenten die gebruik maken van dezelfde serial line uitzetten:

* In “/boot/cmdline.txt” en verwijder alle opties met “ttyAMA0”
* In “/etc/inittab” en commenteer alle opties met “ttyAMA0”

Zie [source code](https://github.com/rubenoosterwijk/rtos_RTLinux/blob/main/serial-tests/serial_uart_test_TxRx.py) voor de code die een bericht verstuurd, leest en valideert.

### KERNEL CODE

Om de globale impact van de verschillende kernel types goed te begrijpen is de volgende post een goede start:

<https://unix.stackexchange.com/questions/553980/why-would-anyone-choose-not-to-use-the-lowlatency-kernel>

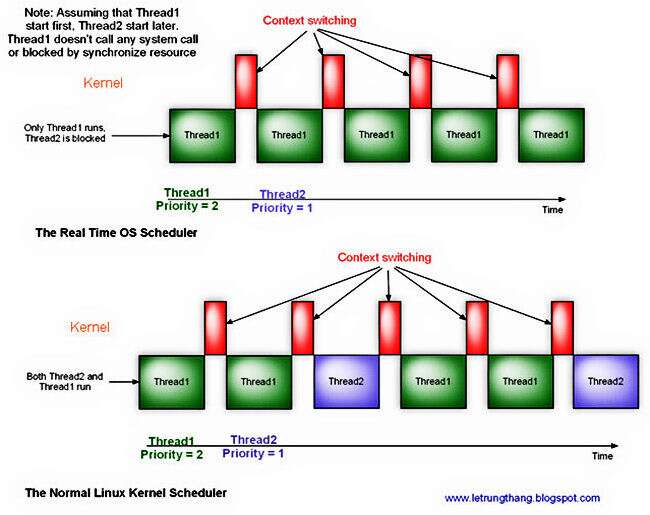
Wat ik wou onderzoeken zijn de verschillen tussen scheduling policies van de standaard linux kernel en de RTOS kernel. Waar ik het volgende artikel als bron heb gebruikt: [bron](https://www.embedded.com/comparing-real-time-scheduling-on-the-linux-kernel-and-an-rtos/).

De normale Linux Kernel is een preventieve kernel, maar natuurlijk niet realtime. In de meeste multithreading-omgevingen (ook wel multitasking genoemd), zorgt een preventieve kernel ervoor dat de thread met een hogere prioriteit langere tijd op de processor kan ontvangen. En omgekeerd heeft een thread met een lagere prioriteit minder tijd voor de processor.

In de normale kernel kan echter geen enkele thread de diensten van de processor de hele tijd monopoliseren, ongeacht de prioriteit ervan. Programma's zullen dus nooit ophangen, zelfs als een willekeurige thread van het programma in een "forever loop" gaat.

Omgekeerd, met bijna elk realtime besturingssysteem (zoals FreeRTOS, Micrium uC/OS en ThreadX ), ondersteunt de kernel zowel preëmptieve als realtime-functies. Dit betekent dat een thread (ook wel een Task genoemd) voor altijd kan draaien als aan de volgende voorwaarden wordt voldaan: (1) hij wordt niet geblokkeerd door een gesynchroniseerde bron (I/O-blok, Mutex, Semaphore...) of (2) hij is niet voorafgegaan door threads die een gelijke of hogere prioriteit kunnen hebben.

Afbeelding 1 hieronder toont de verschillen tussen het threadplanningsbeleid van normale Linux en het threadplanningsbeleid van een RTOS. In figuur 1 nemen we aan dat Thread1 prioriteit 2 heeft en hoger is dan Thread2 dat prioriteit 1 heeft.



Afbeelding 1

In de RTOS-planner wordt Thread1 met een hogere prioriteit altijd uitgevoerd. Thread2 met een lagere prioriteit krijgt nooit de kans om te worden uitgevoerd. Ter vergelijking: de Normal Linux Scheduler doet precies het tegenovergestelde, door zowel Thread1 als Thread2 te laten draaien. Thread1 met een hogere prioriteit heeft een langere looptijd in vergelijking met Thread2 met een lagere prioriteit.

#### Linux-kernel versus RTOS

In een realtime kernelversie van Linux heeft de planner drie (3) planningsbeleidslijnen: Normal , FIFO en Round Robin .

In het normale planningsbeleid wordt een thread gestopt (opgeschort) wanneer een van de volgende drie voorwaarden optreedt:

1. Het wordt geblokkeerd door een toegang krijgende synchronisatiebron (I/O-blok, mutex, semafoon...)
2. Het geeft vrijwillig de controle over de processor op (call sleep() of pthread\_yield() ).
3. De planner onderbreekt de thread wanneer de looptijd is verstreken. De looptijd is afhankelijk van de prioriteit van elke thread, zoals aangegeven in figuur 1.

Het normale realtime planningsbeleid is hetzelfde als het standaard planningsbeleid van de normale kernel, zoals eerder beschreven.

Met het FIFO-planningsbeleid wordt een thread gestopt (opgeschort) wanneer een van de drie voorwaarden zich voordoet:

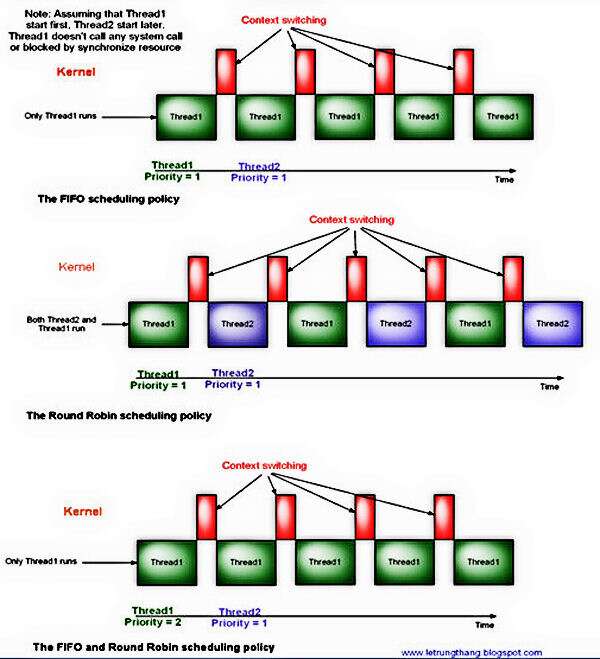
1. Het wordt geblokkeerd door toegang te krijgen tot de synchronisatiebron (I/O-blok, mutex, semafoor...)
2. Het wordt ontkracht door een thread met een hogere prioriteit.
3. Het geeft vrijwillig de controle over de processor op (call sleep() of pthread\_yield() ).

In het planningsbeleid van Round Robin wordt een thread gestopt (opgeschort) wanneer een op de vier volgende situaties zich voordoet:

1. Het wordt geblokkeerd door toegang te krijgen tot de synchronisatiebron (I/O-blok, mutex, semafoor...)
2. Of het wordt ontkracht door een thread met een hogere prioriteit.
3. Of het geeft vrijwillig de controle over de processor op (call sleep() of pthread\_yield() ).
4. Of de tijdschijf is verlopen.

(De meeste RTOS'en integreren het planningsbeleid van Round Robin in hun planner.)

In Afbeelding 2 hieronder hebben we vier situaties: Ten eerste hebben Thread1 en Thread2 dezelfde prioriteit en worden uitgevoerd met behulp van het FIFO-beleid. Ten tweede hebben Thread1 en Thread2 dezelfde prioriteit en worden uitgevoerd volgens het Round Robin-beleid. Ten derde heeft de prioriteit van Thread1 een hogere prioriteit van Thread2 en ze worden uitgevoerd met behulp van het FIFO-beleid. Ten vierde is de prioriteit van Thread1 een hogere prioriteit van Thread2 en worden ze uitgevoerd met behulp van het Round Robin-beleid.



Afbeelding 2

We gaan er ook van uit dat Thread1 eerst start, dan Thread2 later start en tijdens de looptijd doen Thread1 en Thread2 geen systeemaanroepen of worden ze geblokkeerd door een gesynchroniseerde bron (I/O, Mutex, Semaphore...). Het resultaat: de enige verschillen die de realtime prestaties tussen het FIFO- en Round Robin-beleid beïnvloeden, is wanneer Thread1 en Thread2 dezelfde prioriteit hebben.

Conclusie:

In deze context gezien is de typische RTOS-planner slechts een speciaal geval van realtime Linux-planner, of met andere woorden, de RTOS-planner is de realtime Linux-planner die draait met het Round Robin-beleid.